Scheduling Dataflow Execution Across Multiple Accelerators

Jon Currey, Adam Eversole, and Christopher J. Rossbach
Microsoft Research

ABSTRACT
Dataflow execution engines such as MapReduce, DryadLINQ and PTask have enjoyed success because they simplify development for a class of important parallel applications. Expressing the computation as a dataflow graph allows the runtime, and not the programmer, to own problems such as synchronization, data movement and scheduling - leveraging dynamic information to inform strategy and policy in a way that is impossible for a programmer who must work only with a static view. While this vision enjoys considerable intuitive appeal, the degree to which dataflow engines can implement performance profitable policies in the general case remains under-evaluated.

We consider the problem of scheduling in a dataflow engine on a platform with multiple GPUs. In this setting, the cost of moving data from one accelerator to another must be weighed against the benefit of parallelism exposed by performing computations on multiple accelerators. An ideal runtime would automatically discover an optimal or near-optimal partitioning of computation across the available resources with little or no user input. The wealth of dynamic and static information available to a scheduler in this context makes the policy space for scheduling dauntingly large. We present and evaluate a number of approaches to scheduling and partitioning dataflow graphs across available accelerators. We show that simple throughput- or locality-preservation policies do not always do well. For the workloads we consider, an optimal static partitioner operating on a global view of the dataflow graph is an attractive solution - either matching or out-performing a hand-optimized schedule. Application-level knowledge of the graph can be leveraged to achieve best-case performance.

1. INTRODUCTION
This paper addresses the problem of scheduling computations expressed as dataflow graphs on systems which have multiple compute accelerators. We focus on GPU accelerators. Compute fabric of this form is increasingly common: GPU-based super-computers are the norm [1], and cluster-as-service systems like EC2 make systems with potentially many GPUs widely available [2]. Compute-bound algorithms are abundant, even at cluster scale, making acceleration with specialized hardware an attractive and viable approach.

Despite the rapid evolution of front-end programming tools and runtimes for such systems [14, 44, 35, 16, 22, 36, 43, 12], programming them remains a significant challenge. Writing correct code for even a single GPU requires familiarity with different programming and execution models, while writing performant code still generally requires considerable systems- and architectural-level expertise. Developing code that can effectively utilize multiple, potentially diverse accelerators is still very much an expert’s game.

Dataflow execution engines purport to solve many of the challenges introduced by heterogeneity and multiplicity [39, 38, 28, 20]. Because computations are expressed as graphs of vertices representing computation, connected by edges representing communication, a runtime has a complete view of available parallelism, and can implement scheduling and communication policies based on a dynamic view of the system. The separation of concerns simplifies programming, and enables applications to benefit from richer scheduling and communication policies than are available to a programmer with only a static view. Particularly with respect to scheduling, the wealth of dynamic information available to a scheduler does much to reinforce the intuition that automated, optimal or near-optimal scheduling must be attainable. However, our experience building such schedulers [38, 19, 39] suggests that the size of the policy space makes the task of dynamically finding performant scheduling policies a significant challenge, since objective functions may work at cross purposes. For example, a policy that preserves locality by attempting to schedule work where its data are fresh may do so at the cost of poor utilization.

In this paper, we consider a range of scheduling policies in a multi-GPU environment for two real-world dataflow workloads. We find that simple locality- or progress-preserving policies that have been effective in previous systems [38, 39] fail to deliver attractive performance for these workloads, and find that scheduling policies that consider a global view of the graph are necessary for such workloads. We find that while strategies that search for an optimal partition based on a simplified view of the graph achieve the best overall automated performance, outperforming programmer-crafted partitions, application-level knowledge that allows the runtime to duplicate or re-write the graph is necessary to achieve
best-case performance for one of the workloads.

The contributions of this paper are:

1. Proposal, analysis, and evaluations of a range of scheduling policies for dataflow engines based on graph partitioning and graph replication.
2. Evidence that greedy policies based on traditional objective functions such as locality preservation are insufficient in such contexts, and can even result in degraded performance.

2. MOTIVATION

We are motivated in this inquiry by our experience building schedulers for the PTask [38] execution engine. PTask (and by extension EDGE [19] and Dandelion [39]) supports a handful of simple scheduling policies that generally allow the runtime to meet the performance expectations of the programmer. Those expectations are twofold. First, the scheduler should provide good aggregate system-wide throughput in the presence of contention, and second, resources should not go idle when there is a workload that can make performance-profitable use of them. In short, the user expects minimal performance loss under contention, and transparent speedup from multiple accelerators even exists is complicated by the fact that transfer latencies are non-linear in the transfer size. To implement good policy for such workloads, a scheduler must not only consider graph structure but must make accurate predications about dynamic compute and transfer latencies.

In the remainder of this paper, we consider approaches to implementation of a scheduler that can better handle the challenges these workloads illustrate. We examine two production workloads for which we know a greedy scheduler produces poor performance: deep belief neural network training (DBN) for speech recognition, and computing optical flow (OptFlow) on a sequence of video images. Coded for PTask, both workloads are expressed as complex graphs with nested iterative control structures and abundant cycles. We consider a range of designs for which the scheduling algorithm produces a static partition of the graph across multiple accelerators, incorporating increasingly high-fidelity elements in the model on which the partitioner reasons. As these are production workloads, we are also able to evaluate the quality of these partitions against hand-optimized partitions coded by the programmers based on domain and application-level expertise. Our goal is to achieve performance as near as possible to the hand-coded partitions, while minimizing the impact of the partitioner itself, both in terms of compute latency, and in terms of the amount and quality of runtime information required as input.

3. WORKLOADS

3.1 Optical Flow

Optical flow (OptFlow) captures apparent motion of patterns in a sequence of images, and is a common primitive for image processing and computer vision algorithms, e.g. removing rolling shutter wobble from video [9]. Optical flow algorithms are well-studied [10], and generally rely on iterative optimization of an energy function. Our implementation is a variation of the Horn-Schunck [26] algorithm and uses a pyramid of successively smaller scaled versions of input images. Optical flow values are calculated starting from the smallest image versions, using results from each level to...
seed the next level. Each pyramid level features two loops of nested iteration: an inner loop solves a linear system used by each iteration of an outer loop which refines the flow estimate. The PTask expression of optical flow results in a densely connected graph of 126 tasks and 404 edges, amongst which back edges are frequent. Compute latencies and transfer latencies vary significantly with the pyramid level.

3.2 Deep Belief Network Training

The deep belief network (DBN) workload trains a neural network for speech recognition [42]. Like OptFlow, the computation features multiple layers: DBN takes the input samples through a forward pass of the network producing a predicted result. The differences in this result and the ground truth are then used in the back propagation phase to update the weight matrices at each layer through Stochastic Gradient Descent. In its most basic form, neural network training is not parallel because a model must be updated for every input sample. Parallelization strategies trade convergence rate for parallelism, operating on batches of input samples concurrently. The PTask expression of DBN has 65 nodes and 200 edges, with back edges at each layer of the network.

4. DESIGN AND IMPLEMENTATION

The design space for schedulers that operate on a global view of the dataflow graph is large, mostly because the graph model on which the scheduler operates can attempt to capture a number of dynamic metrics at varying levels of fidelity. At one end of the spectrum, brute force search over all possible assignments, based on a model that incorporates a history of measured transfer and compute latencies along with selectivity for predicates on predicated channels in the graph should be able to find an optimal solution that precisely predicts actual runtime behavior. At the other end, a scheduler can limit traversals to local neighborhoods within a graph and employ very coarse performance modes to heuristically identify good graph cuts or sub-graphs - potentially at some performance cost when heuristics fail. In this paper, we attempt to cover important points in the design space with the following policies.

4.1 Heuristic Partitioning

The heuristic partitioning approach favors a minimal compute budget for partitioning algorithms and instrumentation to collect data informing the partitioner. Rather than search the space of assignments, the heuristic partitioner examines local neighborhoods in the graph attempting to identify structures that can be used to estimate whether separating tasks within that neighborhood will yield multiple cut edges elsewhere in the graph. This partitioner considers the density of the connectivity within the local neighborhood, as well as the presence or absence of other runtime level hints such as programmer-configured block pools, which generally are present at points in the graph where communication and allocation latencies are performance-critical. The goal of this heuristic partitioner is to co-schedule logically related sub-graphs without having to traverse the entire graph, and without accepting additional hints from the programmer. While the predictive accuracy of these heuristics varies a great deal, we include this policy because it represents an extreme that strongly favors minimal overhead.

4.2 Optimal Partitioning

The model-based partitioner operates by constructing a weighted model of the graph and applying an optimal partitioning algorithm that finds an assignment across available GPUs that favors utilization (assignments are balanced across all resources) while minimizing communication latency by finding the lowest cost cut that achieves a balanced assignment. We consider both coarse- and fine-grain models of the graph as input to this partitioner. In the coarse case, no attempt is made by the runtime to set edge and vertex weights to reflect actual latencies incurred for communication and computation at those sites in the graph. In the fine-grain case the model weights are set to reflect actual dynamic properties. These two variations represent points in the design space that willingly incur compute overhead for partitioning algorithms, but may or may not invest resources for instrumentation to produce a high fidelity model for the partitioner.

Our current implementation uses an exact combinatorial graph bisection algorithm [21], which is guaranteed to find an optimal solution where one exists. We modify the PTask runtime to generate an abstract model of the graph. In the higher-fidelity variant which models edge and vertex weights, we currently hand-code weights based on our knowledge of the underlying computation. Before putting the graph in the running state, PTask passes the model to an implementation of the optimal bisection algorithm, which determines the assignment of each task to one or other of the partitions. The PTask runtime enforces these assignments using support for mandatory affinity.

4.3 Graph Cloning

The graph cloning approach does not partition the graph, but rather clones it for each available GPU, and partitions the input data across the clones. We include this point in the design space because for some workloads, (streaming, data-parallel, lacking shared mutable state) this is clearly the best policy, and consequently, a scheduler that can correctly identify and exploit the opportunity to apply this strategy is quite attractive. It should be observed that the strategy cannot be safely applied in the general case, since the runtime does not have sufficient information to know when the inputs are completely data-parallel and can be re-ordered arbitrarily. Case in point, this strategy can produce a correct result for the OptFlow workload, but would yield incorrect results if applied to the DBN workload. Evidence of efficacy and general applicability for this policy would likely drive extension of the graph building APIs with primitives that allow the programmer to tell the runtime that this policy is safe for a given graph.

While cloning support can easily be subsumed into the PTask runtime, dividing the input data without some involvement of the application is a greater challenge, since application semantics may determine the points at which the data may be divided.

5. EVALUATION

We evaluate these scheduling policies by measuring their ability to deliver transparent speed-up for the OptFlow and DBN workloads, in the presence of two GPUs. Details of the platform used in these experiments are shown in Table 1 and Figure 1 enumerates the scheduling policies considered. We include the basic FIFO and Data Aware greedy
<table>
<thead>
<tr>
<th>Scheduling Policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO</td>
<td>first-in, first-out</td>
</tr>
<tr>
<td>Data-Aware</td>
<td>favor locality with aged priority</td>
</tr>
<tr>
<td>Hand Partition</td>
<td>hand-optimized</td>
</tr>
<tr>
<td>Heuristic Partition</td>
<td>low-fidelity rule-based</td>
</tr>
<tr>
<td>Optimal Partition</td>
<td>partition using graph model</td>
</tr>
<tr>
<td>Graph Clone</td>
<td>duplicate graph</td>
</tr>
</tbody>
</table>

Figure 1: Evaluated Scheduling Policies

Figure 2: Per-policy speedups on 2 GPUs over execution on 1 GPU for OptFlow

policies present in PTask, along with results based on hand-optimized partitions informed by application-level expertise. We apply graph cloning only to OptFlow, as the transformation does not preserve correctness for DBN.

In the case of optimal partitioning, for OptFlow we explore the use of information about the graph that the application has to hand, and try setting a vertex's weight to its level in the image pyramid (W=L) and to the level squared (W=L^2), in addition to an unweighted model (W=1). For DBN we only try an unweighted model, as the application has no such information to provide. In all cases we use edge weights of 1.

For DBN, the hand-optimized partition provided by the original developers of the workload obtains a non-trivial speedup of 21%. The optimal partitioner is able to improve over it slightly (at 24% speedup) without any domain- or application-level knowledge, or even the assignment of weights in its model.

Combined with the scheduling policy findings in our previous work [38], which profitably applied the greedy Data Aware policy to a range of workloads, there is reason to believe that supporting policies that are effective across diverse workloads is a significant challenge: greedy works for some workloads, partitioning for others, cloning for yet others. We take this as evidence that further exploration of the design and policy space is very much in order.

Specific areas that we plan to investigate include:

- Execution on systems with more than two GPUs.
- Assignment of non-uniform edge weights.
- Derivation of vertex and edge weights from information about historical execution and/or models.
- Strategies for automatic selection of the right high-level approach for a given workload.

6. RELATED WORK

GPUs and Dataflow. StreamIt [47] and DirectShow [33] support graph-based parallelism. OmpSs [15], Hydra [49], and PTask [38], and IDEA [19] all provide a graph-based dataflow programming models for offloading tasks across heterogeneous devices. Liquid Metal [27] and Lime [3] are programming platforms for heterogeneous targets such as systems comprising CPUs and FPGAs. Lime’s filters, and I/O containers allow a computation to be expressed as a pipeline. Flexstream [25] is a compilation framework for synchronous dataflow models that dynamically adapts applications to FPGAs, GPU, or CPU target architectures, and Flexstream applications are represented as a graph. Some of the policies we consider in our evaluation are supported by these systems; to our knowledge none of these systems support the graph partitioning approach we consider here.
Quincy [29] uses (min-cost flow) graph optimization, but not graph partitioning, and applies the concept in a data-center context without attempting to accommodate heterogeneous compute.

Scheduling and Execution engines for heterogeneous processors. Scheduling for heterogeneous systems is an active research area: systems such as PTask [38], Time-Graph [32] and others [48] focus on eliminating destructive performance interference in the presence of GPU sharing. Maestro [41] also shares GPUs but focuses on task decomposition, automatic data transfer, and auto-tuning of dynamic execution parameters in some cases. Sun et al. [45] share GPUs using task queuing. Others focus on making sure that both the CPU and GPU can be shared [31], on sharing CPUs from multiple (heterogeneous) computers [8, 11], or on scheduling on multiple (heterogeneous) CPU cores [13, 7]. Several systems [34, 23] automatically choose whether to send jobs to the GPU or CPU [5, 6, 4, 18, 40], others focus on support for scheduling in the presence of heterogeneity in a cluster [15]. Several systems consider support a MapReduce primitive on GPUs, taking care of scheduling the various tasks and moving data in and out of memory [24, 17, 50]. The same abstraction can be extended to a cluster of machines with CPUs and GPUs [30]. This work was later improved with better scheduling [37]. Teodoro et al. describe a runtime that accelerates the analysis of microscopy image data sets on GPU-CPU clusters [46]. The resulting system relies on task-level dataflow to map the application to a heterogeneous platform, and requires support for cyclic structures in the di-graphs that express the image-processing pipeline. To our knowledge none of these systems support the graph partitioning approach we consider here.

7. CONCLUSION
Our examination of multiple points in the design and policy space suggests that using scheduling to realize speedup transparently from multiple accelerators is a significant challenge, and that a diverse set of policies is likely required to address the needs of diverse workloads. Partitioning based on weighted graph models does not address the needs of all workloads, but it can match or even outperform hand-optimized code for some. Even a fairly coarse model can be effective.

Automatically replicating a graph to exploit parallelism is highly effective in some cases, but is not always safe, and supporting this policy in a runtime therefore requires additional hints from the programmer.

Despite the fact that no clear best policy emerged from this investigation, the existence of performance-profitable policies for these workloads, which frustrated previous PTask schedulers, is reason for hope that the goal is realizable, and the results clearly point to avenues for further research toward it.

8. REFERENCES
2011. ACM.


